

**Architettura degli Elaboratori**  
Secondo semestre

Enrico Bragastini

# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Modello di Von Neumann . . . . .	1
1.1.1	CPU (Central Processing Unit) . . . . .	1
1.1.2	Dal codice di alto livello al binario eseguibile . . . . .	2
1.2	Una CPU didattica . . . . .	2
1.3	Ciclo Fetch-Decode-Execute . . . . .	4
1.4	Architettura della CPU . . . . .	4
1.5	Assembly . . . . .	5
1.5.1	Istruzioni . . . . .	5
1.5.2	Metodi di Indirizzamento . . . . .	5
1.5.3	Microistruzioni . . . . .	6

# Introduzione

## 1.1 Modello di Von Neumann

L'architettura di Von Neumann è una tipologia di architettura hardware per computer digitali programmabili a programma memorizzato la quale condivide i dati del programma e le istruzioni del programma nello stesso spazio di memoria.

Lo schema si basa su 5 componenti fondamentali:

- **CPU** (*Central Processing Unit*), una grande *FSMD* che si divide a sua volta in unità aritmetica e logica (ALU o unità di calcolo) e unità di controllo
- **Memoria**, il luogo dove la CPU recupera istruzioni e dati, contenuti assieme
- **Bus**, il canale di comunicazione tra tutte le componenti dell'architettura. Permette alla CPU di leggere e di scrivere sulla memoria e di acquisire o mostrare dati comunicando con le unità di I/O.
- **Unità di Input/Output**, ovvero i dispositivi di input e output che permettono all'utente di interfacciarsi con la macchina (e viceversa).

### 1.1.1 CPU (Central Processing Unit)

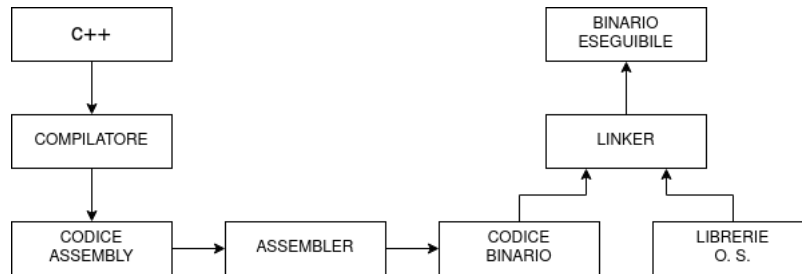
La CPU è l'unità o sottosistema logico e fisico che sovrintende alle funzionalità logiche di elaborazione principali del computer.

Tutte le istruzioni che la CPU è in grado di riconoscere e di eseguire fanno parte del suo **ISA** (*Instruction Set Architecture*), che è in rapporto 1:1 con i codici binari delle istruzioni stesse che vengono eseguite, ad ogni istruzione corrisponde una codifica binaria che il processore è in grado di eseguire. Se due CPU hanno ISA differenti, non potranno eseguire lo stesso sorgente, sarà quindi necessario ricompilarlo separatamente per ogni rispettivo ISA.

In certi casi è possibile che CPU diverse abbiano architetture diverse ma condividano lo stesso ISA. È il caso di *Intel*, che dal processore *80386* ha mantenuto lo stesso set di istruzioni.

### 1.1.2 Dal codice di alto livello al binario eseguibile

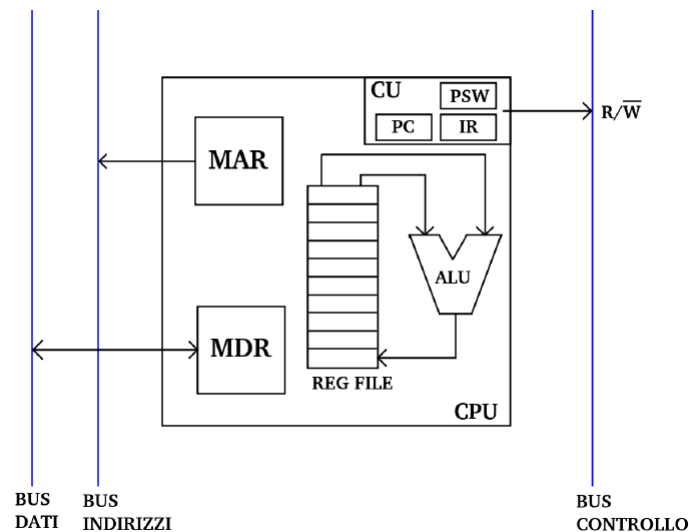
Il processo che avviene quando si compila e si esegue del codice di alto livello, come il C++, è rappresentato nel seguente schema:



Il codice C++ viene compilato dal **compilatore** (GCC in questo caso), il quale produce il relativo **codice assembly**. Questo codice deve essere *assemblato* dall'**assembler** in modo da ottenere del codice binario eseguibile sulla macchina in base alle specifiche dell'*ISA*.

Affinché il codice possa essere eseguito, tenendo a mente che verrà eseguito su una macchina gestita da un Sistema Operativo, è necessario includere i collegamenti alle librerie di sistema richieste dal programmatore in fase di scrittura del codice. A questo proposito interviene il **linker**, che produrrà finalmente un **binario eseguibile**. Il binario prodotto a questo punto è *specifico per quell'ISA* e, per via delle librerie, è anche *specifico per quel Sistema Operativo*.

## 1.2 Una CPU didattica



### Interazione con la memoria

Per interagire con la memoria la CPU ha a disposizione i registri *MAR* e *MDR*, dove comunica i dati da scrivere e dove scriverli. Per la comunicazione effettiva vengono utilizzati 3 diversi BUS.

#### Registri:

- **MAR** (*Memory Address Register*): Contiene l'indirizzo di memoria con cui la CPU vuole interagire
- **MDR** (*Memory Data Register*): Contiene il dato oggetto di scambio, ovvero ciò che viene letto da memoria o che è necessario scrivere sulla memoria

#### Bus:

- **Bus Indirizzi**: Su questo Bus la CPU ci mette gli indirizzi delle celle di memoria con cui vuole operare. La CPU scrive su questo bus e non viceversa.
- **Bus Dati**: Bus adibito al trasferimento dei dati da e verso la memoria. Entrambe la CPU e la memoria leggono e scrivono su questo canale.
- **Bus di Controllo**: Bus in cui la CPU scrive i comandi che devono essere dati alla memoria. I comandi *read* e *write* faranno sapere alla memoria se la CPU ha bisogno di scrivere o di leggere.

### Unità di Controllo

L'unità di controllo **CU** è una *Macchina a Stati Finiti* che gestisce le fasi di ogni operazione. Si occupa di alzare/abbassare tutti i segnali interni di controllo nel momento giusto.

All'interno della CU sono presenti 3 registri specifici:

1. **PC** (*Program Counter*): Registro che contiene l'indirizzo della prossima istruzione da eseguire
2. **IR** (*Instruction Register*): Registro che contiene la *codifica* dell'istruzione corrente da eseguire
3. **PSW** (*Program Status Word*): Registro che contiene una serie di bit specifici, ovvero una serie di informazioni riguardo l'istruzione appena eseguita, utili durante l'esecuzione della successiva.

### ALU (*Arithmetic Logic Unit*)

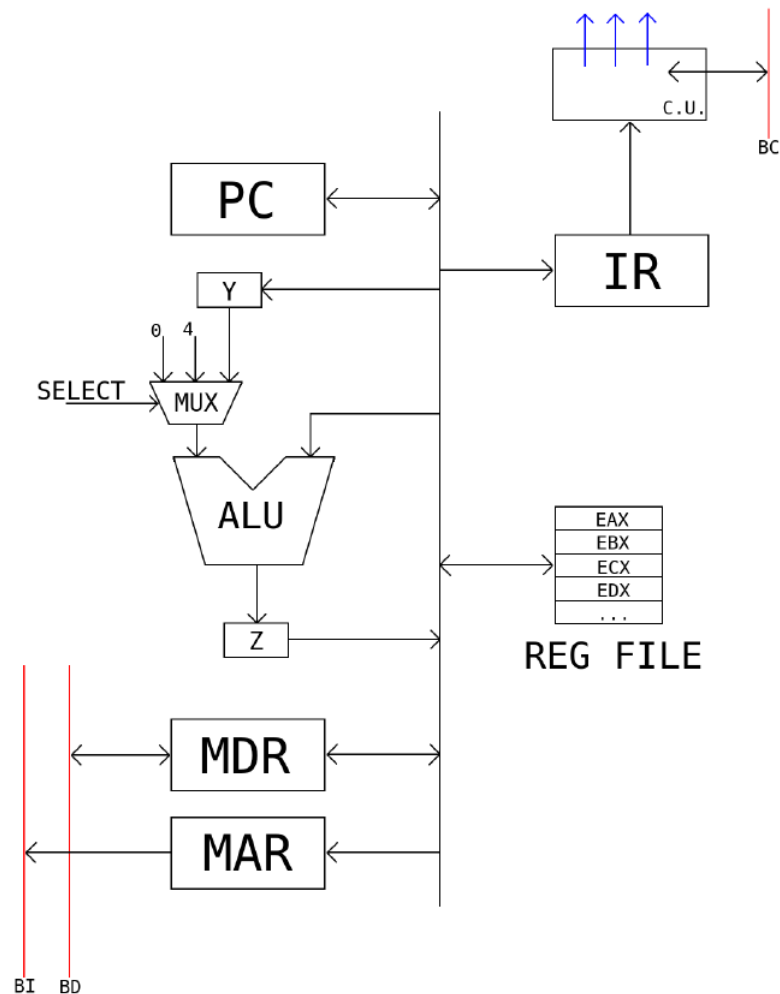
È la sezione della CPU che sa svolgere tutte le operazioni logiche e aritmetiche. È come un grande Datapath controllato dalla sua FSM, ovvero l'Unità di Controllo. La ALU, in base ai comandi della CU, prende i dati dai Registri, esegue l'operazione dovuta e ripone il risultato in un registro.

## 1.3 Ciclo Fetch-Decode-Execute

In termini generali, un processore esegue *iterativamente* **tre operazioni**:

1. **Fetch:** È la fase di caricamento, la CPU carica dalla memoria la prossima istruzione da eseguire. Viene letta la cella memoria il cui indirizzo è contenuto nel *Program Counter*. A questo punto l'istruzione è salvata nell'*Instruction Register* e il PC viene incrementato.
2. **Decode:** È la fase di decodifica dell'istruzione. Viene letto l'Instruction Register per capire cosa è necessario fare. Vengono eventualmente letti anche i valori necessari dalla memoria.
3. **Execute:** È la fase di esecuzione vera e propria dell'istruzione. La ALU esegue i calcoli necessari e i registri vengono modificati.

## 1.4 Architettura della CPU



## 1.5 Assembly

Il linguaggio Assembly è un linguaggio di programmazione molto simile al linguaggio macchina, pur essendo differente rispetto a quest'ultimo.

È concettualmente composto di due parti:

1. **ISA**, ovvero le caratteristiche del linguaggio: le **istruzioni** e i **metodi di indirizzamento**.
2. **Sintassi**, ovvero le regole con cui va scritto il linguaggio.

Il linguaggio Assembly da noi utilizzato sarà l'**Intel 80x86** con sintassi **AT&T**.

### 1.5.1 Istruzioni

Le istruzioni possono avere una **codifica fissa** oppure una **codifica variabile**. A scopo didattico lavoreremo con istruzioni a codifica fissa.

La struttura di un'istruzione è del tipo *OPCODE OPER1, OPER2* dove:

- **OPCODE** è il nome dell'istruzione stessa
- **OPER1** è il primo operando dell'istruzione. In certe operazioni è anche l'unico operando.
- **OPER2** è il secondo operando dell'istruzione. Quando presente, rappresenta anche la *destinazione* dell'operazione, ovvero dove verrà salvato il risultato.

Alcuni esempi di istruzioni:

- MOVL
- PUSHL - POPL
- ADDL - SUBL
- MUL - DIV
- CMPL
- JMP (e derivate)
- CALL
- NOP

### 1.5.2 Metodi di Indirizzamento

Il metodo di indirizzamento dice alla CPU con che modalità deve *recuperare* il dato che le serve per lavorare.

#### Indirizzamento Diretto a Registro

In questo metodo di indirizzamento, il registro viene usato in modo *esplicito*. Basterà quindi usare il nome del registro come operando. Affinché si vada ad operare con lo spazio di memoria del registro stesso

Esempio:            MOVL %EAX, %EBX

### Indirizzamento Immediato

In questo metodo di indirizzamento, una costante viene codificata nell'istruzione stessa, utilizzando il simbolo "\$".

*Attenzione:* una costante non può essere una destinazione e quindi non può essere il secondo operando.

Esempio:            `MOVL $8, %ECX`

### Indirizzamento assoluto

In questo metodo di indirizzamento viene fatto l'uso di **etichette**, la quale rappresenta un indirizzo della memoria con il quale si vuole operare.

Sarà l'*Assembler* ad occuparsi di sostituire il nome dell'etichetta con l'effettivo indirizzo di memoria.

Esempio:            `MOVL DATA, %EBX`

### Indirizzamento Indiretto a Registro

In questo metodo di indirizzamento, si fa utilizzo di un *indirizzo di memoria* contenuto all'interno di un registro. Per accedervi si specifica il nome del registro all'interno di parentesi tonde.

Esempio:            `MOVL (%EAX), %EBX`

### Indirizzamento Indiretto a Registro con Spiazzamento

In questo metodo di indirizzamento, si utilizza sempre un *indirizzo di memoria*, contenuto all'interno di un registro, a cui viene sommata una costante, chiamata **spiazzamento**.

Esempio:            `MOVL $4(%EAX), %EBX`

### 1.5.3 Microistruzioni

Ogni istruzione Assembly viene eseguita in più passi minori. Innanzitutto perché ogni istruzione viene eseguita dalla CPU in tre fasi (*fetch*, *decode* ed *execute*) e ognuna di queste fasi corrisponde a un insieme di **microistruzioni** eseguite internamente dalla CPU.

Esempio di istruzione: `ADDL %EAX, %EBX`

1. Fetch: PCout, MARin, READ, SELECT4, ADD, Zin

Il valore in PC viene "lasciato uscire" sul bus e "fatto entrare" in MAR. Viene dato il segnale di READ alla memoria per leggere la prossima istruzione. SELECT4, ADD e Zin permetteranno PC (gli viene aggiunto il valore 4 mediante la ALU)

2. Fetch: WMFC, Zout, PCin

Viene *messo in pausa* il clock della CPU in attesa che la memoria recuperi il dato e lo scriva sul bus dati.

Inoltre il valore della somma precedente viene fatto uscire dalla ALU e salvato in PC (PC è stato quindi incrementato).



3. Fetch: MDRout, IRin

Il valore dalla memoria è stato scritto sul bus e quindi in MDR. Il dato viene fatto "uscire" da MDR e viene fatto "entrare" in IR. A questo punto l'istruzione da eseguire è stata *caricata* ed è pronta per essere eseguita.

4. Execute: EAXout, Yin

Il valore presente in EAX viene scritto nel registro Y in ingresso alla ALU.

5. Execute: EBXout, SELECTy, ADD, Zin

Il valore in EBX viene fatto "uscire" sul bus e quindi è già leggibile dalla ALU. A questo punto il mux lascia passare il valore di Y nell'altro ingresso della ALU e il segnale ADD fa eseguire la somma tra i due valori. Il risultato viene fatto entrare in Z.

6. Execute: Zout, EBXin, END

Il valore ottenuto dalla somma che si trova in Z viene quindi salvato nel secondo operando, ovvero EBX. Il segnale di END comunica il compimento dell'istruzione.

Essendo i valori da sommare già presenti nei registri, ovvero entrambi con *indirizzamento diretto a registro*, la fase di decode non è necessaria: i valori sono già pronti per essere utilizzati dalla ALU.